Copyright © 1982 American Telephone and Telegraph Company
THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 61, No. 9, November 1982
Printed in U.S.A.

Database Systems:

Database Administration System—Architecture and Design Issues

By C. C. WANG and C. P. HUANG

(Manuscript received September 22, 1982)

The Database Administration System (DBAS) is a software system designed for the Bell Operating Companies to administer several remote, on-line, call-processing-related databases. These remote databases include, for example, the Billing Validation Application files associated with mechanized calling card service, and support for the Automatic Intercept Centers. Briefly, DBAS accepts service-order inputs and forwards them to other databases. DBAS serves as a buffer between the high-speed, real-time-sensitive billing validation applications and low-speed, nonuniform, service-order inputs. DBAS also provides an on-line database to support various administrative functions for the Bell Operating Companies. The major challenge to the DBAS design lies in the size of the database (up to 12-million telephone station records) and its throughput update volume (up to 100,000 random updates per 10-hour day).

I. INTRODUCTION

The Data Base Administration System (DBAS) is a PDP 11/70 computer system under the control of a real-time *UNIX** operating system designed for the Bell Operating Companies (BOCs) to administer several remote, on-line, call-processing-related databases. These remote databases, among others, include the billing validation application (BVA) files located at different network control points for the purpose of providing mechanized calling card service.† Briefly, DBAS

* *UNIX* is a trademark of Bell Laboratories.

† Other remote databases, administered by the DBAS, are not-in-service telephone number data located at the automatic intercept centers and originating (telephone) station treatment data located at the traffic service position systems.

accepts service-order inputs and forwards them to the BVA databases. Because the BVA databases are queried under real-time constraints for processing telephone calls, direct access to them by the BOCs would degrade the performance of the BVAs. DBAS serves as a buffer between the high-speed, real-time-conscious BVAs and the low-speed, nonuniform, service-order inputs. An on-line database at the DBAS site is introduced to further relieve the load of the BVAs from most associated administrative functions: for each telepbone station with data located at a BVA, a DBAS on-line database contains a superset of the data about that station. This superset data includes what is needed for providing mechanized calling card service and much more indirectly related data needed by the BOCs for administrative purposes.

The major challenge of the DBAS design lies in the size of the database and its throughput update volume. A large DBAS database consists of up to 12 million telepbone station records and has the capacity to process up to 100,000 random updates per 10-hour day. These figures are equivalent to (*i*) on-line secondary storage size of close to 1 billion bytes and (*ii*) a limitation to no more than nine disk accesses for an average random record update.

Crash recovery is also very important in the DBAS design. Note that the DBAS database itself is not part of the switching system for call processing, whereas the BVA is. A duplex design of the DBAS database to ensure its high degree of availability would be too expensive because no calls are missed when the DBAS is down. On the other hand, the DBAS database must be available most of the time in spite of system failures because most BOCs plan to operate their DBASs on a six-days-per-week schedule. Since initial loading or reloading of a DBAS database may take from two to five days, the integrity of the database must be maintained at all times so that recovery from a system failure rarely requires reloading the DBAS database.

Given the size of the DBAS database, no existing general-purpose, minicomputer-based database management system (DBMS) satisfies the DBAS update throughput requirement. Clearly, the DBAS application is a special-purpose one. In planning the DBAS architecture, the easy way is to decide what should be included and what should be removed from a typical DBMS to meet the DBAS application. For example, multiple views of the database are supported to provide data independence among different application programs (APs) accessing the database. These APs include clerk input, administrative queries, order processing, order transmitting (to BVAs), and audit (between BVA and DBAS databases). Multiple views allow the flexibility of late binding time among these APs. The interactions among these APs are minimized so that they can be programmed with ease by different people at the same time. Transaction processing is not supported because a

Table I—Characteristics of DBAS

Supporting systems	PDP 11/70
	C language
	Real-time *UNIX** operating system
	UNITY database-management system
Data model	Relational
	Application programs in *C* language
	Access database through function calls
Features	Multiple users
	Multiple user views
	Multiple reads and writes
	Extendible hashing file access
	Variable length records
	Secondary storage management
	Database checkpoint
	Hierarchical locking
	Deadlock prevention
	Secondary key retrieval
	Buffer cache
	Shared segment
	Separate read-only and writable disks
Simplicity in coding and debugging	Message
	User space code
	Synchronous, physical I/O
Capacity and performance	6 RP-06 disks
	12 million records
	3 disk accesses per random record retrieval
	Average 4 to 5 disk accesses per order update
	10,000 order updates per hour
	100,000 records per hour at initial load

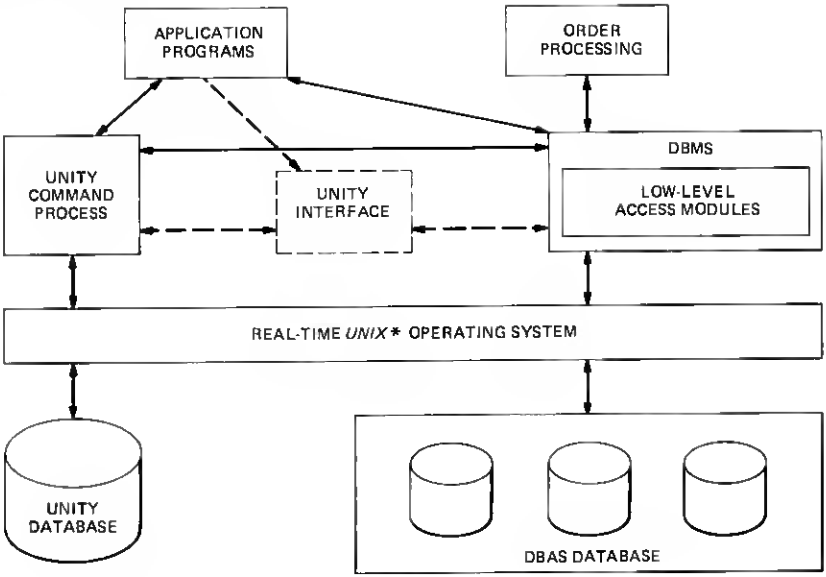
* *UNIX* is a trademark of Bell Laboratories.

"transaction commit" usually requires more disk accesses per record update and this hinders the objective of pushing the high volume of updates through the database. However, a database checkpoint scheme is implemented to facilitate crash recovery. An existing database-management package, UNITY,[1] was adopted for high-level query processing at an early stage of the project so that the available human resources can be directed at designing efficient lower-level access modules. The lower-level modules are directly responsible for meeting the throughput objective. Table I lists the major features of DBAS.

This paper does not cover any of the APs. The overall architecture is described next. Design issues and their solutions by various components of the lower-level access modules are detailed in the remainder of the paper.

II. ARCHITECTURE

From the DBAS database viewpoint, APs fall into one of two categories: (*i*) order processing and (*ii*) administrative report generating and query processing. Order processing is the primary objective of the DBAS in supporting mechanized calling card service. Report generating and query processing serve as the only interfaces between the machine and the BOC administrators. Order processing, which runs all day in



*TRADEMARK OF BELL LABORATORIES

Fig. 1—DBAS architecture.

the background, is disk I/O intensive. Report and query programs, run in the foreground, are not as complex as the queries that would appear in a general-purpose DBMS. Response time is important in most database designs. However, the throughput rate is the main concern in the DBAS design.

The block diagram in Fig. 1 reflects the above perception of the APs. All effort is put in the design of a set of highly efficient lower-level access modules. They are made not only directly accessible to the order processing programs but also to other APs. A random record retrieval and its subsequent update take, on the average, from four to five disk accesses. For high-level processing, a UNITY interface module is planned. The UNITY interface module is to retrieve data from the DBAS database via the lower-level access modules and convert them to the relational format required by the UNITY command modules. The relational operators available in the UNITY command modules are then used for high-level query processing.

2.1 Data models

Telephone stations, their associated equipment and services are the main entities concerning the DBAS database. There are conceptually two types of relations. They are the Billing Number Group (BNG) relation type and the Billing Number Record (BNR) relation type.[2]

These two types are hierarchically related: for each tuple in the BNG type relation there is an instance of the BNR type relation (containing all BNR tuples for that BNG). The tuples of the BNG type and BNR type relations are respectively called the BNG records and BNR records.

A BNR record has the 10-digit telephone station number as its key. The first 6 digits of a 10-digit telephone number identifies a unique BNG record and is also the key of the BNG record. The tuples of a BNR relation represent all active telephone stations with identical six-digit prefixes in their telephone numbers. The lower-level access module does not provide high-level data-manipulation language operations. The basic database access functions provided at the tuple level include (*i*) retrieve, (*ii*) store, (*iii*) replace, and (*iv*) delete a tuple. At the relational level, a complete relation of either the BNG or the BNR type can be retrieved. A complete BNR relation can also be stored or deleted from the database in a single request. A lower-level access module supports a restricted form of predefined views under which an AP may access the database.

2.2 Message and processes

Simplicity in design is the key to the success of a project. We have strived not to duplicate any functions already supported by the *UNIX* operating system unless the throughput objective is at stake. In the area of interface between an AP and a database process, the following constraints are observed to achieve simplicity: (*i*) each AP has at most one outstanding database request and waits while the request is being serviced, (*ii*) a database process services one request at a time and uses no multi-tasking nor asynchronous I/O techniques, and (*iii*) database requests and replies are communicated between an AP and a database process using messages. However, multiple APs can issue database requests to a database process at the same time. They are served in the FIFO order. Even though messages are expensive in terms of machine instructions, their usage minimizes the asynchronous control problem in dealing with multiple database requests from different APs. The handicap of using messages is minimized by passing almost all data among database-related processes through the shared segments.

The DBAS database-management functions are partitioned into more than one process because (*i*) they are too big to fit into the address space of one process and (*ii*) system performance would suffer if both complex low-frequency and simple high-frequency types of database requests from multiple APs were all served by a single database process. The database process would clearly be the bottle neck and would not be able to take full advantage of the multiprogramming services offered by the *UNIX* operating system. The DBAS database processes (Fig. 2)

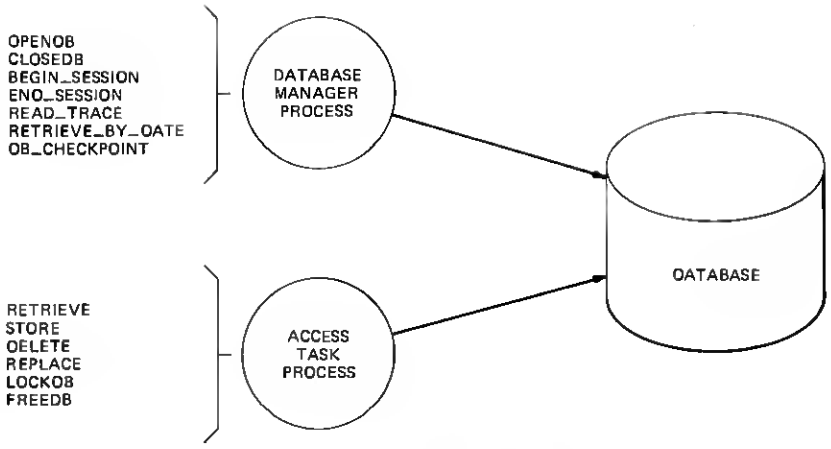


Fig. 2—DBAS Process.

consist of a single Database Manager (DBM) and several identical Access Task Processes (ATPs). The DBM assumes all work that is best suited for a single, centralized process to do. For example, since the free page-address stack is shared and accessed by all database processes for the purpose of allocation and deallocation of disk pages, the jobs of replenishing the stack when it is empty and managing stack overflow are the sole responsibility of the DBM. Specifically, the DBM sets up and initializes most of the data structures required in the shared segments, semaphores variables for use in dealing with critical sections, and processes database checkpoints. The ATPs are restricted to moving data in and out of the database according to requests by APs. The binding of an AP's view and its server ATP occurs at the database view opening time. The benefit is, again, in simplicity of design for no dynamic server scheduler is needed in offering multiple ATPs.

2.3 Data-manipulation primitives

Table II lists a set of data-manipulation primitives for an AP to interact with database processes. They are implemented as a set of standard library functions residing in each AP's address space. Messages sent to and received from the database processes are embedded in these routines and therefore transparent to the APs. The primitives, OPENDB and CLOSEDB, also connect and disconnect the shared segment in the AP's address space when necessary. When a view is opened by an AP, the DBM allocates a system work area in the shared segment. For each database request, the work done by the corresponding routine on the AP side includes moving data between the AP's

Table II—Data manipulation primitives

DBM	ATP
OPENDB	RETRIEVE*
CLOSEDB	STORE*
BEGIN_SESSION	DELETE*
END_SESSION	REPLACE
READ_TRACE	LOCKDB
RETRIEVE_BY_DATE	FREEDB

* The amount of data is either a tuple or a relation.

workspace and a specific system work area in the shared segment, and sending a message to and waiting for a reply from the database processes.

The amount of data passing through the system work area is one tuple at a time. The format of the tuple, including the ordering and data types of the fields in the tuple, is according to an AP's view. An entire relation, in a restricted form, can also be retrieved in a single retrieve command: in this case, the bulk of the data is passing from the ATP to the requesting AP in a file.

2.4 Shared segments

The database processes rely heavily on the shared segments supported by the real-time *UNIX* operating system. They are used as storage for common data, as means to save core space, and to get around the limitation of small virtual address space imposed by the machine. A shared segment is also used in moving data between an AP and database processes.

Segments used in DBAS are named (*i*) AP-segment, (*ii*) ATP-segment, and (*iii*) buffer-segment. The AP-segment is shared among the DBM, all ATPs, and all APs accessing the database. For each view opened by an AP, a unique system work area in the AP-segment is allocated for the purpose of moving data between the AP and ATPs. The ATP-segment is internal to the database processes and not shared with the APs. Data structures, shared internally among the database processes, include, among others, the free-page address stacks, the top level of the database tree, and concurrency control structures. Most structures on the AP-segment and ATP-segment are dynamically allocated and freed. Routines of the *UNIX* operating system are modified for the purpose of managing the individual segment space.

Buffer caches are implemented through buffer-segments. A connected segment always occupies the same address space within an ATP so that address pointers within a segment are always meaningful. An ATP can connect to only one buffer-segment. There are more buffer-segments than there are ATPs. A buffer-segment, disconnected from an ATP and saved at the end of a retrieval-type database request of an

order update, is most likely reconnected by the ATP to process the subsequent replacement type database request from the same AP. This type of buffer cache eliminates three disk accesses per order update (i.e. per-paired retrieval and replacement-type database requests).

III. DESIGN AND IMPLEMENTATION ISSUES

3.1 File structure and file access

The DBAS application and its throughput requirement impose the following constraints on the database file structure design.

- A file large enough to deal with up to 12 million records.
- Records of variable length. They can be dynamically inserted and deleted.
- Access of a random record that requires as few disk accesses as possible.
- Facility to retrieve all BNG records and all BNR records of a given office code, npa-nxx.

The reasons that the file and directory structures of the real-time *UNIX* operating system can not meet the needs are discussed in Section 3.4. Among the other candidates for the choice of a suitable file structure, the B-tree[3] and extendible hashing methods[4,5] require the fewest number of disk accesses in retrieving or storing a random BNR record out of a population of 12 million. An order update accesses both a BNR record and its hierarchically related BNG record. The interesting problem is how one structures the BNG records in an efficient manner such that they do not cost additional disk accesses in an order update. The solution used in DBAS is a two-stage extendible hashing algorithm. The structure of the data (Fig. 3) is essentially a

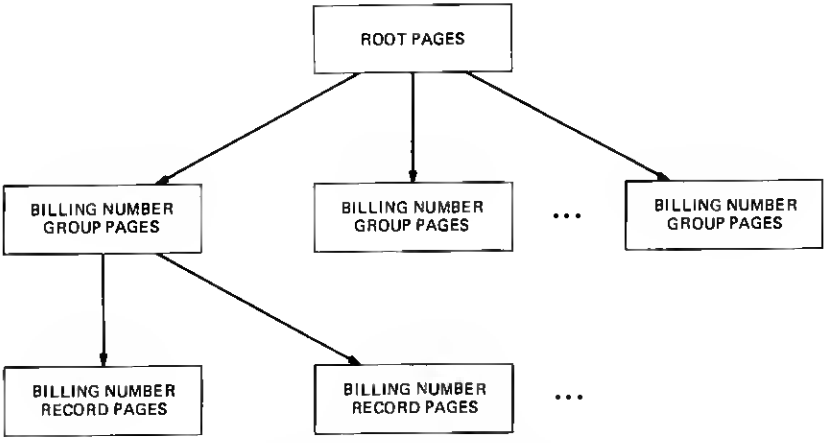


Fig. 3—Primary database structure.

two-level hierarchical one consisting of BNG records of an operating telephone company at the first level, and BNR records (of individual billing numbers) at level two. Extendible hashing is used at both levels to access respectively the BNG record and the BNR record.

3.1.1 Extendible hashing algorithm

Briefly, given a key, the algorithm first computes its hashed key value (hkv), and then makes use of the hkv to search a binary tree, stored in bit-vector form, to compute a logical page address (la). The la is a mask of the low-order bits of the hkv. The size of the mask (number of bits) depends upon the level of the node in the binary tree. From la and a logical address-to-physical address map (LPAM), the corresponding physical page address (pa) is known. The disk page at pa is read into the memory buffer, and the records within the buffer are sequentially searched until the one with the matched key value is found.

The process of storing a record follows the same steps as searching a record in bringing a page into the memory buffer. Room for the new record within the unused area of the (page) buffer is allocated. The new record is then moved into its assigned area of the buffer and the buffer content is written out onto disk to complete the insertion operation. In case the page brought into memory does not have enough room for the new record, a new page is allocated. The records within the page just brought in and the new record are distributed among the two pages according to their hashed key values. The two pages are each assigned a new logical page address according to the hashed key values of the records it contains. The bit vector, which represents the relationship of all defined logical page addresses, is consequently updated. Similarly, the entries corresponding to the affected logical page addresses within the LPAM are also updated.

3.1.2 Data structures used in extendible hashing

The following types of input data structures are needed for each application of the extendible hashing algorithm:

- A bit vector, its height and size.
- The LPAM pages. The size of the LPAM varies dynamically. It can occupy several disk pages. Its size grows and shrinks more easily if it is not required to occupy consecutive physical page addresses.

3.2 Concurrency

Concurrent operations are necessary for the DBAS to achieve its throughput objectives. At the hardware level, multiple disk controllers are used to maximize the data-path bandwidth between the disks and

the main memory. At the software level, multiple APs can access the database at the same time. Multiple copies of the ATP are simultaneously running to serve the APs to minimize the CPU idle time.

Locks are used to solve the data-conflict problem. In order to simplify the implementation of any locking scheme, locks on logical records are assumed to be translated into equivalent locks on the corresponding physical pages. The locks are chosen so that (*i*) tbey are easy to use and to design, and (*ii*) they provide a high degree of concurrency. The following considerations are immediately noted.

3.2.1 Logical and physical locks

A race condition exists when two different APs attempt to update the same data at the same time. The negative consequence of a race condition is that the database may no longer be consistent. Data conflict may occur at either the logical record level when APs access the same BNG record (or the same BNR record), or the physical record level when APs access different BNG records (or BNR records) that happen to be located on the same physical page. Since physical pages are transparent to the APs, locking on physical pages takes place implicitly when an AP explicitly locks a logical record.

3.2.2 Lock granularity

Clearly, the smaller the sizes of lock granules are, the higher is the degree of concurrency that can be achieved. However, locking of a BNR record, the smallest logical record in the DBAS database, necessitates the implicit locking of several physical pages. There is therefore a direct link between the lock granularity and complexity in implementing it.

3.2.3 The location of a lock

The locks can be kept in the main memory or on the disk next to where the locked data item is. The shortcomings of keeping them on disk include (*i*) more disk accesses are required in accessing the locks, and (*ii*) the data items locked by an aborted AP may become permanently inaccessible.

3.2.4 Deadlock

When an AP is allowed to issue more than one lock, there is the probability of deadlock occurring.[6] Locks left by aborted APs may also introduce the deadlock problem.

The DBAS locking scheme is a much simplified version of the hierarchical locks described in Ref. 7. An AP can either lock the entire database or a BNG record for exclusive access. When a BNG record is locked, the associated physical page containing the BNG record is

implicitly locked. Because of the size of the database, the probability of two random BNG records residing on the same page is much less than one hundredth. Even though the size of the lock granule is grossly large, the degree of concurrency is adequate for DBAS applications. Moreover, the distinction between logical and physical locking is practically eliminated from the implementation. An AP is permitted to own at most one lock at any time so that deadlock due to multiple active APs waiting for one another can be prevented. Locks left behind by aborted APs are cleared periodically at about 5-minute intervals to avoid causing the system to wait indefinitely.

Data structures for the locks and their corresponding queues are kept in the main memory. Specifically, since they have to be accessed by all the ATPs, they are located in the shared ATP-segment. Lock and free lock can be either stand-alone database requests or piggy-backed to other types of database requests such as retrieve and replace to minimize the message overhead in using them for order processing.

3.3 Database checkpoint

The DBAS database must be reliable. It does not need to be operational every minute. However, it should not be down for extended periods such that the system cannot clear its backlog of updates. A database update operation comprises, in general, several disk writes. A database transaction is commonly defined as a sequence of updates that transforms the database from one consistent state to another consistent state. Because minimizing the number of disk accesses per update is the main concern in the DBAS design, and adopting database transactions to handle updates would have incurred more disk accesses, DBAS does not have the concept of a database transaction. It follows that the database may become inconsistent if the system fails in the middle of an update. The most undesirable way to restore the consistency of the database is to reload the database. Since it takes more than three days to reload an average DBAS database, the consistency of the database must be maintained under the condition of system failure so that reloading becomes unnecessary. This is solved by performing periodic database checkpoints. At each database checkpoint, a consistent copy of the database is saved on disks. When the system is restarted after a failure, the most recent consistent copy, saved at the last database checkpoint before the failure, is used, and reloading of the database is avoided at the expense of losing the updates entered between the last checkpoint and system failure.

As a further precaution to confine the catastrophe due to fatal disk I/O errors to a small region, we partition the database disks into read-only disks and writable disks. The read-only disks contain the most consistent copy of the database at the end of each day. The writable

disks are initially empty at the beginning of the day, and contain an increasingly larger portion of the database as the day progresses. Fig. 4 illustrates the update effects on the writable disks. At the end of the day, the contents of the writable disks are all merged onto the read-only disks to give a new consistent copy of the database. The probability of a hardware write-protected disk drive having a fatal I/O error is much smaller than that of a drive permitting both read and write. Consequently, almost all fatal disk I/O errors are confined to the writable disk drives. The work lost due to a fatal disk I/O error is therefore limited to one day's work of updates. A duplex system was also considered for DBAS database and rejected because of its cost.

3.4 Secondary storage management

3.4.1 Introduction

The real-time *UNIX* operating system supports both contiguous and noncontiguous files. The noncontiguous file has the advantage that the management of free pages is part of the file system function. However, the number of disk accesses to retrieve a page of a large (noncontiguous) file is too many for the DBAS application. In the case of contiguous files, the file system uses the concept of multiple extents to provide the capability of file growth and shrinkage. All these advantages can be fully utilized if the sizes of the files are not bigger than the size of the host file system. The major difficulty lies in the restriction that the size of a file system can not exceed the capacity of a special device (174 million bytes in the case of RP-06). In other words, a file system in the *UNIX* operating system does not span more than one disk drive. The DBAS database needs a file of size much larger than one RP-06 disk. This large file would have to be artificially partitioned into multiple file systems if one insisted on using those provided by the *UNIX* operating system. The negative impacts include (*i*) unusually large number of file systems have to be mounted at the same time when the mount points are already scarce resources in most systems, (*ii*) an unusual number of files would have to be introduced to take advantage of the space-management facility of the *UNIX* operating system, and (*iii*) more file-open and file-close operations would have to take place due to the limitation on the number of files that are allowed to stay open at any time for each process. These negative impacts obviously affect the DBAS throughput objective.

The following considerations are noted in the design of a special Secondary Storage Management module (SSM) for DBAS use. (See Table III for a list of SSM features.)

3.4.1.1 Transparency of multiple database disks. The SSM should make the distinction between multiple database disks and a single database disk transparent to the file-access method. On the other hand, the SSM

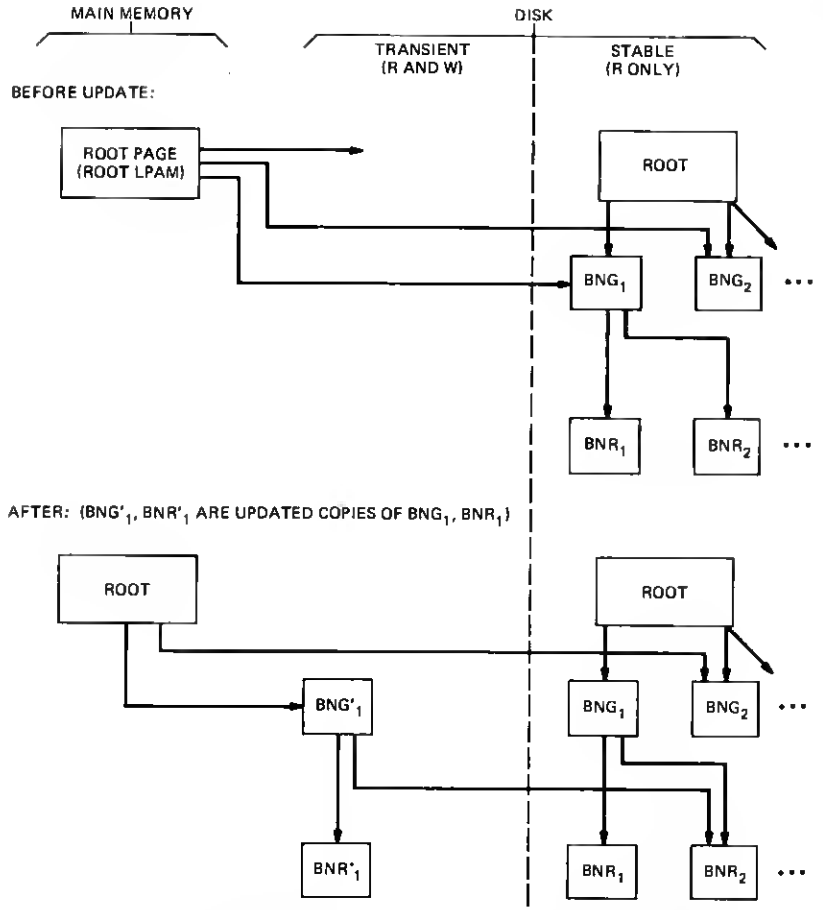


Fig. 4—Illustration of the update effects on the writable disk.

should separate the read-only disks from the writable disks such that data on the read-only disks can take advantage of the hardware write protection feature.

3.4.1.2 Efficiency. Allocation and deallocation of a block should require almost no disk accesses. Reading (or writing) a block from (or to) a disk takes exactly one disk access. Data movement in reading or writing a block should also be minimized.

3.4.1.3 Contiguous disk space. Reading from (or writing to) successive disk blocks requires fewer disk seeks than from (or to) disjoint blocks. In general, the number of I/O calls is the same as the number of blocks being moved from (or to) the disk. The SSM can provide a further improvement in efficiency by issuing exactly one I/O call in moving multiple blocks when the source and the destination are known to

Table III—Secondary storage-management features

Up to 6 RP-06 disks
2K-byte disk blocks
Write 1 to 4 contiguous blocks in 1 system call
Separation of read-only and writable disks
Contiguous free-space management
Fragmented free-space management
Data placement heuristics
Support for database checkpoint
Support for multiple copies of ATPS

occupy contiguous space. Since the DBAS updates are confined to the writable working disk during the day and merged to the read-only disk each night, all blocks on the working disk are free in the morning when the system starts. Moreover, all read-only disk blocks are also, obviously, free when an initial database load starts. Careful management of contiguous space cuts down daily update time and the long period required for database load.

3.4.1.4 Fragmented disk space. The choices of data structures for managing fragmented disk space include, among others, the following.

(*i*) A free-page bit vector. The position and the binary value of a bit is used to represent, respectively, one page and its allocation status. One bit vector is required for each database disk.

(*ii*) External free-page address file. A file, external to the database disk space, is used to record all the free-page addresses.

(*iii*) Internal free-page address file. A linked list of blocks, where each block is a part of the database disk space, is used to record all the free-page addresses.

The first two data structures occupy space that can otherwise be allocated for other purposes. The third one, similar to the structure used in most cases to manage the main memory free blocks, occupies the spaces that are free and unused due to fragmentation.

3.4.1.5 Checkpoint support. When modifying a block, if the old content is needed for the purpose of restart after a system crash, then the old block cannot be over-written, and a new shadow block must be used for the storage of the modified content. Even though the contents of the old block are obsolete, the deallocation of the old block for reuse must be deferred until the shadow block is checkpointed. The SSM should manage the deferred free blocks efficiently to support the database checkpoint.

3.4.1.6 Data placement heuristics. The BNG records in the DBAS database are obviously accessed thousands of times more frequently than the BNR records. The BNG record should be placed in a special disk area so their accesses cause the least amount of head movements.

Other considerations in the SSM design include 2K (2048) byte block size, disk reconfiguration for database growth, and migration.

3.4.2 Implementation

The picture of the database disks, presented to the file-access method by the SSM, is a large, virtual, contiguous file with multiple extents of usable and nonusable space. Each of the multiple extents of usable space corresponds to a database disk. The blocks, each having 2K bytes, of this large virtual file are addressed by 32-bit numbers. The first five of the 32 bits is used to address a database disk and the remaining bits to address the block offset within the disk.

The multiple extents of usable space of this large virtual file are further partitioned into three types of areas.

(*i*) BNG area. A set of cylinders close to the center of a read-only disk is dedicated for the storage of BNG related record. Since the BNG records are presumed to be accessed at a high traffic rate, the space of the remaining areas of the same disk is allocated in a discrete manner so that there is always a high probability that the disk head stays within the BNG area at all times.

(*ii*) BNR area. With the exception of the BNG area, all other areas on the read-only disks are used to store BNR records and other miscellaneous data.

(*iii*) Working volume area. The areas on the writable disks are for the purpose of update processing.

Each area of the three types has contiguous unused space and fragmented free space. A contiguous unused space is simply managed by keeping its size, and its lower- and upper-bound block addresses in the system. The fragmented space of an area type is managed through the aforementioned internal free-page address file. The names of the three (internal) free-page address files are GFL, RFL, and WFL for respectively the BNG, BNR, and working volume area types. Furthermore, to achieve efficiency during the allocation and deallocation of the blocks of each area type so that no extra disk accesses are incurred, a free-page address stack is maintained for each area type. They are replenished from, or overflowed to, their respective free-page address files when nearly empty or full.

The management of deferred free pages in supporting database checkpoints is similar to that of fragmented free pages of the working volume area. A fourth (internal) free-page address file, called DFL, and a companion free-page address stack are allocated and operated the same way in managing the deferred free pages.

Finally, when multiple copies of the ATPs are running, they all need to access the SSM data structures. These data structures are all placed on the commonly shared ATP-segment and accessed through the use of semaphores to avoid any critical section problems from occurring.

3.5 Buffer management

A set of page buffers is statically allocated within the user address

space to form a buffer pool. A block of data that is moved in and out of the database must first be placed in one of these buffers. They are time shared among the different page types of the database. The objectives are to minimize the amount of data movement and to facilitate implementation of the extendible hashing algorithm. The following considerations are noted in the buffer management design.

3.5.1 Number of buffers

The data space of 64K bytes allowed to each process on the PDP 11/70 sets an implied upper limit on the amount of space that can be allocated to buffers. The internal operation of the database requires page splitting, which can demand 4 to 6 pages to be buffered simultaneously.

3.5.2 Contiguity of multiple buffers

Contiguous free space on disk is used to reduce the number of disk I/O operations in updating a record. This requires that at least some set of buffers in the pool occupy contiguous memory space.

3.5.3 Buffer cache

Updating requires two accesses to the same record. A cache buffer arrangement is implemented to keep the record in memory between these two accesses, while allowing the ATP to act on other requests. With this arrangement nearly 50 percent of accesses will find the record already in memory. This hit ratio is even higher when a large number of sequential records are updated, since they will usually go to the same page.

Each buffer has 2K bytes. An ATP has a pool of six buffers in its address space, four of the six occupying a contiguous 8K shared buffer-segment. Buffer-segments are used to provide the buffer cache capability. A buffer-segment stack, located on the ATP-segment, is used to manage the free buffer-segments.

To support database checkpoints, the SSM time-stamps each page when it is written on disk. When a block is to be copied from a memory buffer to a disk, the time stamp decides whether the old disk block can or cannot be overwritten. If the time stamp is earlier than the last checkpoint time, then it must be saved for the purpose of restart after a system crash, and a new disk block, onto which the modified buffer contents are copied, must be allocated. Clearly, writing the buffer contents to a new disk block may affect pointers in other buffers. The accurate processing of this chain reaction is part of the objective of the buffer descriptor array.

The state of the buffers in the buffer pool is described by a buffer-descriptor array. Each entry of this array records whether the buffer

is free or used, disk block addresses from which and to which the contents of the buffer is copied, and the relationship of the buffer contents to other buffers.

3.6 Secondary key retrieval

A service order containing a future service-effective date is called a pending order. Because of the size of the DBAS database, it is impossible to scan through the entire database to find all the BNR records with a given effective service date. The DBAS database management system provides a restricted secondary key retrieval capability for the purpose of processing and administering pending orders. Given a (future) service-effective date, the retrieval-by-date function lists the keys (telephone station number) of all BNR records that contain the given (service effective) date.

Since the retrieval-by-date function is not expected to be used very often, the main concern is to devise a structure so that its maintenance during a regular order update will not incur extra disk accesses. An inverted list is chosen for the secondary key retrieval. During normal updates, new entries of the inverted list are piled on top of the old ones in a dedicated main memory buffer similar to a free-page address stack. Whenever the buffer is full, its contents are moved to a disk block and linked to the rest of the inverted list.

When processing a retrieval-by-date request, the inverted list is sorted and filtered to produce the results. Obsolete entries of the inverted list are removed nightly during database merge time.

3.7 Database load

3.7.1 Introduction

The initial loading of the DBAS database from magnetic tapes prepared by an BOCS data centers is an expensive, time-consuming process. For a large database, it takes four days to load the database from scratch to its full size. If regular updates were used to insert one record at a time to the database, the loading time would be at least ten times longer (e.g. 40 days). The main goal of the DBAS database initial-load program is to shorten the total database loading time. Features used to achieve this goal include (*i*) a linear depth-first search algorithm[8] to avoid repeated writing of the same disk blocks, (*ii*) taking advantage of the contiguous disk space to reduce the number of disk writes, (*iii*) options of running multiple copies of the database initial-load program to increase throughput rate, and (*iv*) a checkpoint to minimize the degree of work loss due to system crash. The following linear depth-first search algorithm—which stores all BNR records of a given BNG record in the database—illustrates where contiguous disk space is used to reduce the number of system calls.

3.7.2 Store-all algorithm

Assume that the records R1, R2, ..., Rn are prearranged in ascending hashed key values K1<K2<...<Kn. Initially, the binary tree, T, consists of only the root. The current buffer, CB, is empty and is assigned the logical page address 0 at current level L = 1.

1. [Iteration]. For $i=1,2,...,n$ do steps 2 to 7. At the end of this iteration, a clean-up operation is done to complete the loading.

2. [Input]. Read record Ri.

3. [Output current buffer to disk?] Compute the logical page address Pi and level Li from Ki and current binary tree T. If Pi is the same as the current buffer logical page address, do step 4. Otherwise, the content of CB is stablized. If CB is non-empty, it is written onto a disk. A new CB is allocated and initialized, and its logical page address is set to Pi. The current level L of CB is set to Li.

4. [Add record to current buffer]. If the current buffer has room for the in-coming record Ri, place Ri on the current buffer and go to step 2 to process the next record. Otherwise, since the current buffer CB is too full for record Ri, do steps 5 to 7.

5. [Allocate an additional buffer for tree splitting]. Allocate a buffer, CB′.

6. [Grow tree T by splitting contents of CB at level L]. Split the contents of the full buffer CB at level L between CB and CB′ at level L+1: The logical page address of CB and CB′ at level L+1 are LCHILD(Pi, L), and RCHILD(Pi,L). The logical page address of each record of the full buffer CB at level L is recomputed, if it agrees with the new logical page address of CB at level L+1, it remains in CB. Otherwise it is put in CB′.

7. [Select the new current buffer]. Compute the new logical page address Pi from Ki and the newly split tree T. If Pi agrees with that of CB, keep CB as the current buffer. In this case, CB′ must be empty, and CB is still too full for record Ri. Set L to L+1 and repeat step 6. On the other hand, if it agrees with that of CB′, then the content of CB is stabilized. If CB is non-empty, it is written out onto disk. Let the new current buffer CB be CB′, set L to L+1 and repeat step 4.

Note 1. By writing out a buffer contents immediately after it becomes stabilized, the description of the algorithm is simplified. In reality, buffer contents are not written out until no more contiguous buffers can be allocated. Contents of contiguous buffers are moved to consecutive pages on disk in a single system call. This reduces not only the number of seeks from (up to) 4 to 1, but also economizes the CPU usage during initial load.

Note 2. The cleanup operation moves what remains in the buffer pool to disk. These buffers include the contiguous ones that contain records not yet stored on disk, buffers used to build the logical-to-

physical address translation, and the buffer containing the BNG record with the newly constructed binary tree in bit-vector form.

Note 3. The left child node has logical page address LCHILD(Pi, L) = Pi. The right child node has the logical page address RCHILD(Pi,L) = 2**L + Pi.

IV. Experiences

DBAS, being a vital link between an BOC and the BVAS at network control points in supporting the nation-wide mechanized calling card service, is a production database management system. Most BOCs in the Bell System have committed themselves to installing DBAS by second quarter, 1982. The turnover of the DBAS Generic 2DB3 to its first customer, Southwestern Bell in St. Louis, Missouri, was right on schedule in June 1981. An early Generic, 2DB2, was also on schedule in its delivery to New York Telephone in July 1980. The data, collected so far from the field, show that the design has very successfully met its capacity and throughput rate objectives.

The following are some of the contributing factors towards the success of the project.

(*i*) The DBAS design team is throughput conscious and goal oriented. The time and coding complexities of each component have been closely monitored throughout the design and implementation stages.

(*ii*) The DBAS database-management modules comprise approximately 30-thousand lines of the C language code. Its modular and layered structure has made the debugging and trouble-shooting tasks manageable.

(*iii*) The interaction among the application programs is minimized through multiple user views (supported by DBAS). The programmers, developing the APS, do not have to go through cumbersome and error-prone tasks in negotiating a common data header among themselves in the process of designing and debugging their individual programs. The benefit is also apparent in the shorter overall system testing and integration time.

(*iv*) The evolutionary approach—getting the essential programs to work first and partitioning the entire job into two stages (Generics 2DB2 and 2DB3)—has the function of boosting the confidence and morale of the developers and other members of the DBAS project in delivering the products on schedule.

V. ACKNOWLEDGMENTS

The authors wish to acknowledge the contributions from the other members of the DBAS design team. In particular, D. A. Dixon implemented the user interface modules and played the role of a database administrator in incorporating all APS' views into the DBAS database;

G. M. Jensen implemented an earlier version of the secondary storage management modules; and J. E. Simpson assisted in the preparation of this paper. Finally, the authors wish to thank the testing and integration team for its efforts in debugging the system and the management team for its continuing trust during the course of this development.

REFERENCES

1. D. T. Chai and P. D. Ting, "UNITY—A Microcomputer DBMS Stand-alone and Distributed Environment," The 1979 IEEE Electronics Conference, New York City, April 24–26, 1979.
2. J. W. Schmidt, "Some High Level Language Constructs for Data of Type Relation," ACM Trans. Database Systems, *2*, No. 3 (September 1977), pp. 247–261.
3. D. E. Knuth, *The Art of Computer Programming: Sorting and Searching*, Reading, Mass.: Addison-Wesley, 1972.
4. R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible Hashing—A Fast Access Method for Dynamic Files," ACM Trans. Database Systems, *4*, No. 3 (September 1979), pp. 315–344.
5. P. Larson, "Dynamic Hashing," BIT, *18*, No. 2 (1978), pp. 184–201.
6. A. C. Shaw, *Logic Design of Operating Systems*, Englewood, New Jersey: Prentice-Hall, 1974.
7. J. N. Gray, "Notes on Data Base Operating Systems," in *Operating Systems: An Advanced Course*, Vol. 60, *Lecture Notes in Computer Science*, New York: Spring-Verlag, 1978.
8. A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Reading, Mass.: Addison-Wesley, 1975.